Structured bindings

Document Number: **P0144R2**
Date: 2016-03-16
Reply-to: Herb Sutter (hsutter@microsoft.com), Bjarne Stroustrup (bjarne@stroustrup.com), Gabriel Dos Reis (gdr@microsoft.com)
Audience: EWG

Abstract

This paper proposes the ability to store a value and bind names to its components, along the lines of:

```
tuple<T1,T2,T3> f(/*...*/) {  /*...*/  return {a,b,c}; }
auto [x,y,z] = f();  // x has type T1, y has type T2, z has type T3
```

This addresses the requests for support of returning multiple values, which has become a popular request lately. Proposed wording appears in a separate paper, **P0217**.

Contents

0. Revision history

R2: Applied feedback from Jacksonville: Switched from `auto {x,y,z}` syntax to `auto [x,y,z]`. The wording supports bitfields. Added `tuple_size` check. Expanded `get<>` to use `tuple_element<>`.

R1: Applied feedback from Kona, including added support for `get<>` to let a type opt into binding.

1. Motivation

Today, we allow multiple return values via `tuple` pretty nicely in function declarations and definitions:

```
tuple<T1,T2,T3> f(/*...*/) {     // nice declaration syntax
    T1 a{}; T2 b{}; T3 c{};
    return {a,b,c};              // nice return syntax
}
```

We enable nice syntax at the call site too, if you have existing variables:

```
T1 x; T2 y; T3 z;
tie(x,y,z) = f();                // nice call syntax, with existing variables
```

However, this has several drawbacks:

- It works only for separately declared variables.
- If those variables are of POD type, they may be uninitialized. This may violate reasonable coding rules.
- If they are non-PODs or initialized PODs, they may be initialized redundantly – first to a placeholder or default value (possibly using default construction) and then again to their intended value.
- Even default construction is often undesirable, for example if `f()` is an attempt to open a file stream and return the stream together with an outcome status. Being able to declare and initialize the variables at the same time would be much more direct and more natural to read.

What we could like is a syntax to store a value and introduce names for its components at the same time:

```
declare-and-tie(x,y,z) = f();    // nice call syntax, to declare and initialize
```

2. Proposal

We propose extending the local declaration syntax to allow a single declaration that stores a value with individual names for its components. We support the cases when the value:

- is an array, or
- supports access via `get<N>()` with matching `tuple_size` (incl. `std::tuple` and `std::pair`), or
- has all public non-static data members.

2.1 Basic syntax

For the basic syntax, we want to make the new form distinct enough from the current form that requires the variables to have the same type (e.g., `auto x = 1, y = 2, z = 3;`, or `auto x, y, z = f();` which initializes only `z` today).

There are several unused syntaxes available that we could use to express this case. Per EWG feedback in Kona and Jacksonville, this paper will pursue this basic syntax:

```
auto [x,y,z] = f();              // brackets
```

because it is more visually distinct from the existing syntax for declaring multiple variables of the same type.

2.2 Basic model

The basic model is to put the value returned by `f()` into a local variable (whose lifetime lasts until the end of the scope, as usual) and make the introduced names be lvalues referring to the corresponding components. The number of elements or members must equal the number of names.

The declaration

```
auto [x,y,z] = expression;
```

stores the value of *expression* and makes the individual names `x`, `y`, and `z` available to refer to the components.

Let `N` be the number of introduced names (`1` or more), and `E` be the type of *expression*. There are three cases, considered in order. Note: These are pseudocode approximations showing references, but this does not imply an actual reference; in many (probably most) cases the compiler can eliminate the references and access elements of the result directly.

Case 1, built-in array: If `E` is an array type, then the individual names correspond to the array elements. The following is pseudocode only.

```
auto __a = expression;

auto& x = __a[0];
auto& y = __a[1];
auto& z = __a[2];
```

Case 2, get<>: If `std::tuple_size<E>::value` is well-formed and evaluates to `N`, the individual names behave as having type `tuple_element<#,decltype(E)>` and values as given by from `get<#>(expression)` where `#` is `0` for the first name up to `N-1` for the last name. The following is pseudocode only.

```
auto __a = expression;

tuple_element<0, decltype(E)>::type& x = get<0>(__a);
tuple_element<1, decltype(E)>::type& y = get<1>(__a);
tuple_element<2, decltype(E)>::type& z = get<2>(__a);
```

A `get<>` that needs access to nonpublic data members would need to be a friend or be implemented in terms of a suitable member or friend function.

Case 3, public data: Otherwise, if all of `E`'s non-static data members are public and are declared in the same base class of `E` (`E` is considered a base class of itself for this purpose), then the individual names are bound using a declaration-order traversal of the non-static data members of `E`. `E` must have the same number of data members as the number of introduced names (in this example, three), and no union members. The following is pseudocode only.

```
auto __a = expression;

auto& x = __a.mem1;        // does not imply an actual reference
auto& y = __a.mem2;
auto& z = __a.mem3;
```

Notes: `std::tuple` and `std::array` fall into case 2. C-style structs and `std::pair` fall into case 3.

For example:

```
tuple<T1,T2,T3> f();
auto [x,y,z] = f();                          // types are: T1, T2, T3

map<int,string> mymap;
auto [iter, success] = mymap.insert(value);  // types are: iterator, bool

struct mystruct { int i; string s; double d; };
mystruct s = { 1, “xyzzy”s, 3.14 };
auto [x,y,z] = s;                            // types are: int, string, double
```

2.3 Customization

The reason to put case 2 before case 3 is to permit customization for structures. For example, given:

```
struct S {
    int i;
    char c[27];
    double d;
};

S f();
auto [ n, s, val ] = f();
```

What if the author of `S` wants `s` to be a `string`? This proposal does not currently support writing “`string`” on the `s` parameter (see Q&A 3.6)

```
auto [ n, string s, val ] = f();             // NOT proposed
```

on the ground that this complicates the proposal and might block the path to pattern matching. However, by putting case 2 first we can provide a set of getters if that is suitable:

```
// add tuple_size and tuple_element support

namespace std {
    template<> struct tuple_element<0,S> { using type = int;    };
    template<> struct tuple_element<1,S> { using type = string; };
    template<> struct tuple_element<2,S> { using type = double; };

    template<> struct tuple_size<S>: public integral_constant<size_t,3> {};
}

template<int> void   get(const S&)    = delete;

template<>    auto get<0>(const S& x) { return x.i;         }
template<>    auto get<1>(const S& x) { return string{c,i}; }
template<>    auto get<2>(const S& x) { return x.d;         }

auto [ n, s, val ] = f();                    // now s is a string
```

2.4 Qualifying `auto` with a *cv-qualifier* or reference

Any cv-qualifiers or &/&& used to qualify `auto` apply to `__a`. For example, the declaration

```
auto const& [x,y,z] = expression;
```

is as in 2.3 but with `__a` declared as

```
auto const& __a = expression;
// ... x, y, and z as appropriate for the type of expression ...
```

2.5 Range-`for`

The syntax is also available in range-`for`. For example:

```
map<widget, gadget> mymap;
for(const auto& [key,value] : mymap)        // read-only loop
    { ... }
```

This makes it simpler to deal with things like key/value pairs while avoiding the longstanding pitfall of forgetting where to put the `const` to avoid an unintended conversion.

2.6 Move-only types

Move-only types are supported. For example:

```
struct S { int i; unique_ptr<widget> w; };
S f() { return {0, make_unique<widget>()}; }
auto [ my_i, my_w ] = f();
```

3. Q&A: Other options/extensions considered

3.1 Should this syntax support initialization from an `initializer_list<T>`?

We think the answer has to be no, primarily because the size of an `initializer_list` is dynamic whereas the list of introduced names is static.

3.2 Should this syntax support initialization from a *braced-init-list*?

For example:

```
auto [x,y,z] = {1, "xyzzy"s, 3.14159}; // NOT proposed
```

We think the answer should be no. This would be trivial to add, but should be well motivated and we know of no use cases where this offers additional expressive power not already available (and with greater clarity) using individual variable declarations. This can always be proposed separately later as a pure extension if desired.

3.3 Should we also allow a non-declaration syntax without `auto` to replace `tie()` syntax?

For example:

```
[x,y,z] = f();                          // same as: tie(x,y,z) = ...
[iter, success] = mymap.insert(value); // same as: tie(iter,success) = ...
```

We think the answer should be "no, at least for now." We know of no use cases where this is better than using `std::tie`, as noted in the comments. This can always be proposed separately later as a pure extension if desired.

3.4 Should qualifying `auto` with `&&` be supported?

Yes, mainly because of range-for. Note that `auto&&` is a "forwarding reference," which is usually for parameters. The one notable valid local forwarding use is to forward a value to a range-`for` loop body:

```
for( auto&& [first,second] : mymap ) { // proposed
    // use first and second
}
```

3.5 Should the syntax be extended to allow `const`/`&`-qualifying individual names' types?

For example:

```
auto [& x, const y, const& z] = f();   // NOT proposed
```

We think the answer should be no. This is a simple feature to store a value and bind names to its components, not to declare multiple variables. Allowing such qualification would be feature creep, extending the feature to be something different, namely a way to declare multiple variables.

If we do want to declare multiple variables, we already have a way to spell it:

```
auto val = f();                         // or auto&&
T1&       x = get<0>(val);
T2 const  y = get<1>(val);
T3 const& z = get<2>(val);
```

3.6 Should this syntax support concrete types?

For example:

```
Payload [x,y] = f();                    // NOT proposed: same type
auto [x, string y] = f();               // NOT proposed: conversion to string
```

We think the answer should be no.

For the first case, note that the `Payload` example can already be written as:

```
auto [x,y] = Payload{f()};
```

For the second case, this is a simple feature to store a value and bind names to its components, not to declare multiple variables. Allowing concrete types would be feature creep, extending the feature to be something different, namely a way to declare multiple variables. In the discussions on the reflectors and in Kona, the `string` example was repeatedly mentioned as a reason for allowing explicit specification of a type for individual variables. The suggested source would be some kind of C-style string that needed conversion to string. Note that this conversion is easily achieved using a `get<N>` function as shown in section 2.3.

3.7 Should this syntax support concepts?

For example:

```
Iterator [x,y] = f();                  // NOT proposed: same concept
something [Iterator it, bool b] = f(); // NOT proposed: different concepts
```

We think the answer should be no. This is a simple feature to store a value and bind names to its components, not to declare multiple variables. Allowing specific concepts or types would be feature creep, extending the feature to be something different, namely a way to declare multiple variables.

If we do want to declare multiple variables, we already have a way to spell it with the Concepts TS extensions:

```
auto val = f();
Iterator it = get<0>(val);
bool     b  = get<1>(val);
```

3.8 Should there be a way to explicitly ignore components?

The motivation would be to silence compiler warnings about unused names.

We think the answer should be "not yet." This is not motivated by use cases (silencing compiler warnings is a motivation, but it is not a use case per se), and is best left until we can revisit this in the context of a more general pattern matching proposal where this should fall out as a special case.

Symmetry with `std::tie` would suggest using something like a `std::ignore`:

```
tuple<T1,T2,T3> f();
auto [x, std::ignore, z] = f();  // NOT proposed: ignore second element
```

However, this feels awkward.

Anticipating pattern matching in the language could suggest a wildcard like _ or *, but since we do not yet have pattern matching it is premature to pick a syntax that we know will be compatible. This is a pure extension that can wait to be considered with pattern matching.

3.9 Should there be support for recursive destructuring?

For example:

```
std::tuple<T1, std::pair<T2, T3>, T4> f();
auto [ w, [x, y], z ] = f();           // NOT proposed: types are T1, T2, T3, T4
```

We think the answer should be "not yet." This could be a future extension, following experience with the basic feature and in languages like Python.

Acknowledgments

Thanks to Matt Austern, Aaron Ballman, Matt Calabrese, Chandler Carruth, Jonathan Caves, Tom Honermann, Nevin Liber, Jens Maurer, Gor Nishanov, Thorsten Ottosen, Richard Smith, Oleg Smolsky, Andrew Tomazos, Tony Van Eerd, Ville Voutilainen and everyone else who contributed feedback and discussion on drafts of this paper.