Structured bindings

Document Number: **P0144R1**
Date: 2016-02-03
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 declare multiple variables initialized from a `tuple` or struct, 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

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 declare and initialize variables:

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

2. Proposal

We propose extending the local variable declaration syntax to allow:

- a single declaration that declares one or more local variables,
- that can have different types (and so must have a distinct syntax from the current multiple variable declaration syntax where all variables have the same type),
- whose types are always deduced (using a single `auto`, possibly cv-qualified or `&`-qualified),
- when assigned from a type that supports access via `get<N>()` (including `std::tuple` and `std::pair`) or whose non-static data members are all public.

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 the proposal in revision R0 and subsequent EWG feedback in Kona, this paper will pursue this basic syntax:

```
auto {x,y,z} = f();              // braces
```

because it is more visually distinct from the existing syntax for declaring multiple variables of the same type, and the `{x,y}` introduction syntax is used in concepts.

2.2 Direct and copy initialization

This paper proposes allowing direct and copy initialization, such as:

```
auto { list-of-comma-separated-variable-names } { expression };
auto { list-of-comma-separated-variable-names } = expression;
```

For simplicity the examples will focus on copy initialization, but all examples apply to the various forms.

2.3 Basic type deduction

The declaration

```
auto {x,y,z} = expression;
```

declares the variables `x`, `y`, and `z`, and deduces their respective types and initial values from *expression*.

Let `N` be the number of declared variables (`1` or more), and `E` be the type of *expression*. There are three cases, considered in order:

Case 1, built-in array: As with range-`for`, if `E` is an array type, then the declared variables' types and initial values are deduced from the types and values of the array elements as if we had written the following by hand:

```
auto&& __e = expression;

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

Case 2, get<>: As with range-`for`, if a member `.get<#>()` or else a nonmember ADL-visible `get<#>()` exists and can be invoked with `__e` defined as `auto&& __e = expression;` and values of `#` from `0` to `N-1`, then the declared variables' types and initial values are deduced from the types and values of `get<#>(expression)` where `#` is `0` for the variable declared first up to `N-1` for the variable declared last. Initialization of the declared variables is performed as if we had written the following by hand (including for lookup of the calls to `get`):

```
auto&& __e = expression;

auto x = get<0>(__e);           // nonmember case
auto y = get<1>(__e);
auto z = get<2>(__e);
```

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 declared variables are initialized from a declaration-order traversal of the non-static data members of `E`, and their types are deduced from their initializers as if individually declared `auto`. `E` must have the same number of data members as the number of variables declared (in this example, three), and no union members.

Notes:

- To be efficient, the wording is crafted to allow implementation latitude such as copy elision.
- `std::tuple` and `std::array` fall into case 2, and 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
```

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 we want `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:

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

template<>    auto   get<0>(const S& x) { return x.i; }
template<>    string 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*

As with individual variable declarations, here `auto` can be cv-qualified. The declaration

```
auto const {x,y,z} = f();                    // const T1, const T2, const T3
```

is as in 2.3 but declares `x`, `y`, and `z` to be `const`.

2.5 Qualifying `auto` with &

As with individual variable declarations, here `auto` can be `&`-qualified. If the initializer is an rvalue, then it must also be `const`-qualified, and results in lifetime extension of the returned `tuple`. Thus the declaration

```
auto const& {x,y,z} = f();                   // const T1&, const T2&, const T3&
```

is as in 2.3 but declares `x`, `y`, and `z` to be `const&`.

Further, the declaration

```
auto& {x,y,z} = f();                          // ERROR, illegal for an rvalue
```

is not legal for a returned rvalue, but can be correctly written by turning the rvalue into an lvalue:

```
auto const& val = f();                 // or just plain "auto" to copy by value
auto& {x,y,z} = val;                   // ok, initializer is an lvalue
```

2.6 Range-for

The syntax is also available when declaring variables 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.

2.7 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 variables to be defined 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. It would also complicate the grammar because `{` is already permitted in

this position to begin a block, so we would need lookahead to disambiguate. (Using `()` parens is worse, because code like `(iter, success) = expression;` already has a meaning and in some cases might compile today.) 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 variables' types?

For example:

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

We think the answer should be no. This is a simple feature intended to bind simple names to a structure's components by value or by reference. We should avoid complication and keep the simple defaults simple. (If we want `const`/`&` qualification, why not a concept/type name too? Those should be considered together; see 3.6.)

We already have a way to spell the above, which also makes any lifetime extension explicit:

```
auto const& val = f();                // or just plain "auto" to copy by value
T1&       x = get<0>(val);
T2 const  y = get<1>(val);
T3 const& z = get<2>(val);
```

Secondarily, we could be creating subtle lifetime surprises when the initializer is an rvalue:

- Should a single `const&` extend the lifetime of the whole `tuple`? The answer should probably be yes, but then this could cause surprises by silently extending lifetimes for the other values in the tuple.
- Should the use of non-const `&` be allowed? If we allow any `const&` to extend lifetime, then non-const `&` would also be safe as long as there was some other variable being declared using `const&`. But that would be inconsistent with the basic case, and create quirky declaration interactions.
- Should only `const`, but not `&`, be allowed? That would avoid the above problems, but feels arbitrary.

3.6 Should this syntax support non-deduced (concrete) type(s)?

For example:

```
string {x,y} = f();                   // NOT proposed: same type
auto {x, string y} = f();             // NOT proposed: conversion to string
```

We think the answer should be no. This is a simple feature intended to bind simple names to a structure's components by value or by reference. We should avoid complication and keep the simple defaults simple. Complication here might block evolution paths to a more general form of pattern matching.

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 intended to bind simple names to a structure's components by value or by reference. We should avoid complication and keep the simple defaults simple.

As noted in 3.5, we already have ways to spell all of the above. For example, with the Concepts TS extensions we can already write:

```
auto&& val = f();                        // or just plain "auto" to copy by value
Iterator it = get<0>(val);
bool     b  = get<1>(val);
```

The argument could be made that in other cases we do not have mandatory type deduction, so that for parameters and concepts you can choose among: specifying the exact type (no deduction); specifying a concept (constrained deduction); and specifying `auto` (unconstrained deduction). If we do not allow that here, we are breaking consistency for return value binding. We don't think that argument holds, because the purpose of this feature is to supply a simple default that you can already write out by hand. We do not need to burden a targeted feature with ornamentation to become a second (and ornate) way to spell something we can spell already; indeed, that would be missing the point of adding a simple default.

Finally, allowing specific concepts or types would be feature creep. The purpose of the feature is to bind new names to the values that are already there. Even replacing `auto` with a concept or type as in the first group of lines changes the meaning, because in the proposal `auto{x,y}` can deduce different types whereas presumably the concept or type must apply to all variables. This will nearly always be wrong, and people will want to mention the concept and type names on the individual components, leading right back to 3.5 with the same observations noted there, including that we already have a way to spell those things.

3.8 Should there be a way to explicitly ignore variables?

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

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`, which uses `std::ignore` in expression, would suggest using something like a `std::ignore_t` (since this is a declaration, not an expression):

```
tuple<T1,T2,T3> f();
auto {x, std::ignore_t, 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, Jonathan Caves, Tom Honermann, Nevin Liber, Jens Maurer, Gor Nishanov, Thorsten Ottosen, Richard Smith, Oleg Smolsky, Andrew Tomazos, Tony Van Eerd, and Ville Voutilainen for feedback and discussion on drafts of this paper.