# Proposal of Multi-Declarators (aka Structured bindings)

Document No.: P0151R0
Project: Programming Language C++ - Evolution
Hostile-Revision-Of: P0144R0
Author: Andrew Tomazos <andrewtomazos@gmail.com>
Date: 16 Oct, 2015



## Summary

We propose a better approach for “Structured Bindings” as defined in P0144R0 - where “better” is defined as terser, more ortogonal, more general, more expressive, less (parsing-wise) ambiguous. The opening example in P0144R0 is:

```
tuple<T1, T2, T3> f();

auto{x,y,z} = f();  // P0144
```

Some examples of our syntax, from the most verbose to most terse, are:

```
tuple<T1, T2, T3><T1 x, T2 y, T3 z> = f(); // P0151 most verbose
tuple<T1, T2, T3><x, y, z> = f(); // P0151 implicit auto on members
<T1 x, T2 y, T3 z> = f(); // P0151 implicit auto on return value
<x,y,z> = f(); // P0151 most terse (implicit auto on both)
```

# Background / Definitions

We define *multi-initialization*, as a feature whereby multiple variables are declared and initialized by a single shared “compound” initializer. (This has also been called “structured bindings”, “tied declaration” and “multi return/assign”.)

Recall that the direct binding of a reference to an object is also a form of initialization. Recall that a variable is either an object or a reference.

In general multi-initialization is syntactic sugar for an existing long notation where a “compound” variable is initialized with a single initializer, and then individual variables are initialized by values decomposed from it.

That is, given some initializer **expr**:

First, we want to use **expr** to initialize an invented temporary variable **t** of type **T**.

Then, we want to initialize variables **m1** through **mn** of type **M1** through **Mn**, with initializers that refer to the “sequence-decomposition” of t: **t1, t2, …, tn:**

```
T t = expr;
M1 m1 = t1;
M2 m2 = t2;
…
M3 m3 = tn;
```

We define that some types have a *sequence decomposition*. Given a value *t* of a type T with a sequence decomposition, the sequence decomposition of t is a fixed length sequence of values, the lifetime of which are enclosed by that of t. (Usually this means they will be subobjects of t, but not always). Standard types with a sequence decomposition include at least std::pair, std::tuple, std::array and simple structs (details tbd). User-defined types may also be defined to be sequence-decomposable by providing some supporting entities (details tbd). Notably the length of the decomposition must be a function of the type T (and not depend on the value t. For example, std::vector does not have a sequence decomposition).

# Proposal

The first proposed change add a new kind of declarator called a *multi-declarator.* A multi-declarator is enclosed by angle-brackets (to disambiguate it from other syntactic

constructs that can appear in a simple declaration). A multi-declarator allows us to collapse away **t** and the **ti** initializers and make the above sequence decomposition implicit:

```
T <M1 m1, M2 m2, …, Mn mn> = expr;
```

(Because T is a type, a following `<` unambiguously introduces a multi-initialization statement.)

Next we notice that each occurrence of T, M1, M2 thru Mn can be individually replaced with a concept or by auto:

```
auto <auto m1, auto m2, …, auto mn> = expr;
```

## Implicit Auto

We propose that each occurrence of auto as a decl-specifier in a declaration statement that contains a multi-declarator can be individually omitted:

```
<m1, m2, …, mn> = expr;
```

## Unnamed Declarators

Further we propose that within a multi-declarator a declarator-id can be omitted (like in a function parameter declaration list):

```
<int i, float /*r*/, string s> = f();
```

## Copy Elision

We also propose that given the variable t is not visible except through the sequence decomposition, that any copies should be available for copy elision. That is, if a temporary is created to hold t, and a copy construction is made from some ti to initialize mi, that copy construction may be elided (and so mi will just be a reference to the subobject within t).

# Corollaries

We explore some of the consequences of the proposal. We assume “std::get<I>” is adapted to provide sequence decomposition.

## Corollary 1

Given:

```
std::tuple<int, float, string> f();
```

The long form:

```
std::tuple<int, float, string> t = f();
int i = get<0>(t);
float r = get<1>(t);
string s =get<2>(t);
```

is equivalent to each of the following declaration statements:

```
std::tuple<int, float, string><int i, float r, string s> = f();

auto<int i, float r, string s> = f();

<int i, float r, string s> = f();

std::tuple<int,float,string><i, r, s> = f();

auto<auto i, auto r, auto s> = f();

<i, r, s> = f();

<int i, auto r, s> = f();

std::tuple<int,float,string><int i, auto r, s> = f();
```

## Corollary 2

Given:

```
const std::tuple<int, float, string>& f();
```

The long form:

```
const std::tuple<int, float, string>& t = f();
int i = get<0>(t);
float r = get<1>(t);
```

```
const string& s =get<2>(t);
```

is equivalent to each of the following declaration statements:

```
const std::tuple<int, float, string>&
                <int i, float r, const string& s> = f();

const auto&<int i, float r, const string& s> = f();

const& <int i, float r, const string& s> = f();

const& <i, r, const& s> = f();
```

## Corollary 3

Multi-declarators, like most compound declarators, can be nested:

Given:

```
struct Point { int row, col; };
struct Rect { Point topleft, botright; };

Rect get_rect();
```

The long form:

```
Rect t1 = f();
Point t2 = t1.topleft;
Point t3 = t1.botright;
int top = t2.row;
int left = t2.col;
int bot = t3.row;
int right = t3.col;
```

is equivalent to each of the following declaration statements:

```
Rect<Point<int top,int left>,Point<int bot,int right>> =
get_rect();

auto<auto<auto top, auto left>, auto<auto bot, auto right>> =
   get_rect();

<<int top, int left>, <int bot, int right>> = get_rect();
```

```
<<top, left>, <bot, right>> = get_rect();
```

## Corollary 4

Given:

```
std::map<float, float> ageweights;
```

The long form:

```
for (auto ageweight_keyval : ageweights) {
  float age = ageweight_keyval.first;
  float weight = ageweight_keyval.second;
  ...
}
```

Is equivalent to each of:

```
for (<float age, float weight> : ageweights) ...

for (<age, weight> : ageweights) ...
```

# Wording Sketch

Add new production “multi-declarator” to noptr-declarator:

noptr-declarator:
 declarator-id
 **multi-declarator**
 noptr-declarator parameters-and-qualifiers
 noptr-declarator [ constant-expression]
 ( ptr-declarator )

**multi-declarator:**
 **< parameter-declaration-list >**

Give a declaration statement the semantics (as described under the proposal) as being equivalent to a set of invented declaration statements, the first of which initializes a local

variable of a unique name and then use the sequence decomposition of that as the initializers in the rest of the declaration statements.

# Comparison with P0144

The proposed syntax in P0144:

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

suffers two key problems:

First, “auto {x, y, z}” is nearly ambiguous with a braced-init-list initialized temporary, and hence “auto {x, y, z} = f()” is nearly ambiguous with an expression statement that is an assignment statement. Certainly if we replace auto with a concrete type T, it is ambiguous. We find this ambiguity unacceptable.

Second, the P0144 proposed syntax entails mandatory type deduction. It doesn’t allow concrete types (or concepts) to be specified for either the type T, or the types M1, M2, …, Mn. Numerous members have made it clear that this is unacceptable. In every other similar construct in the language the spectrum of type deduction can be applied from concrete types to the “any type” concept auto.

In contrast, the syntax proposed in this proposal is an elegant and orthogonal extension to the existing declaration statement syntax, parses unambiguously, supports the full spectrum of concrete types/concepts/type-deduction for **both** the temporary t **and** the resultant m1, m2, …, mn, and furthermore, the most terse form is even terser (this has been achieved because an opening `<` unambiguous introduces a multi-declarator declaration statement. `{`, on the otherhand, would introduce a compound-statement):

```
auto<x, y, z> = f(); // same as P0144
<x, y, z> = f();  // same thing, even terser than P0144
```

We therefore assert that the proposed syntax in this proposal is strictly better in every way than that of P0144.