Document #: N4074
Date: 2014-07-02
Reply to: Herb Sutter
hsutter@microsoft.com

Let *return {expr}* Be Explicit, Revision 2

Contents

History and Acknowledgments

This paper is a followup to N4029 to follow EWG preference for a variation on that proposal, namely allowing explicit constructors to be called by the syntax `return {expr};`, with the same semantics as initializing a local variable using *`return_type`* `var{expr};`. This addresses EWG issue #114.

Thanks to Richard Smith for his assistance with the proposed wording. Thanks to Peter Dimov, Gabriel Dos Reis, Howard Hinnant, Stephan T. Lavavej, Mike Miller, Bjarne Stroustrup, Ville Voutilainen, Jonathan Wakely and all others who provided comments on this topic and/or drafts of this paper.

Discussion

C++ already recognizes that the expression in a `return` statement is special. For example, for `return x;` where `x` is the name of a local variable, we now consistently treat the expression as an rvalue and so can move from it. This makes perfect sense, because clearly we're not going to use the variable again anyway since we're returning—what else could we possibly have wanted?

For `return {expr};`, this paper proposes permitting explicit conversions from `expr` to the return type. (Note: This paper does not propose changing the meaning of return statements without a braced-init-list.)

After all, in the case of a function declared to return a value of type (possibly cv-qualified) T:

- `return {expr};` inherently means to use `{expr}` to initialize the returned T object; furthermore, it uses initialization list syntax which again is explicitly for initialization. This is de facto an 'explicit' and 'direct' initialization syntax because it cannot mean anything else. What else could we have wanted but to directly and explicitly initialize that T object? Forcing the user to repeat the type with `return my_long::type<name>{expr};` does not add value, and is redundant.
- The function's `return` statement and return type are both owned by the same person, the function author/maintainer.

The status quo is also arguably inconsistent with initialization of locals whose type is specified. Given:

```
struct Type1 {          Type1(int){} };
struct Type2 { explicit Type2(int){} };
```

The statement `return` *`expr-that-evaluates-to-int`*`;` works only if the ctor is not `explicit`, which leads to the following inconsistencies:

```
Type1 f1() {
    Type1 local1{1}; // ok
    return {1};      // ok
}

Type2 f2() {
    Type2 local2{2}; // ok
    return {2};      // error, message could be "you must write Type2{2}"
}
```

I believe this is inconsistent because the named return type is just as "explicit" a type as a named local automatic variable type. Requiring the user to express the type is by definition redundant—there is no other type it could be.

This is falls into the (arguably most) hated category of C++ compiler diagnostics: "I know exactly what you meant. My error message even tells you exactly what you must type. But I will make *you* type it."

Although the proposal was originally motivated by returning a `tuple`, which is being addressed by other library proposals, the motivation goes beyond `tuple`. Any other changes made to `tuple` are orthogonal to this proposal.

Some Key Viewpoint Questions

In the discussions so far, one's position on any given example as a problem or non-problem often seems to depend on one's view of a small set of fundamental questions—irrespective of the specific example.

1. Is a return type conceptually "local" or "distant" with respect to the return statement?

Specifically, when X is a type with an explicit constructor from `expr`, we have:

```
X f() {
    X local{expr};     // A: ok
    return {expr};     // B: error today
}
```

In discussions about specific X's and `f`'s, the actual X's and `f`'s tend to disappear after a while and concerns always seem to boil down to whether lines A and B are both "local." For line A, I have seen no one suggest that allowing explicit constructors to fire is somehow problematic. For line B:

- Those who view the return type as "local" to the function ("whoever is authoring and maintaining the function also knows and maintains the return type") seem to universally like this proposal, and want line B to be legal.
- Those who view the return type as "distant" from the return statement (e.g., "it could be up off the screen, or was written by the original author but maybe not remembered well by the maintainer") seem to universally dislike this proposal, and want line B to continue to be an error.

This paper argues that the return type is local. If you are authoring or maintaining the function, that includes the return type. As Bjarne put it in Portland, if you don't know the return type of your function, something bigger is amiss.

One last argument: If you believe that the return type is distant and want B to continue to be an error, then please consider that we trivially refactor all the time back and forth between the following two pieces of code, which just extracts or unextracts a return variable:

```
X f() {
    ...
    ...
    ...
    return {expr};     // B: error today
}

X f() {
    X ret{expr};       // A: ok
    ...
    ...
    ...
    return ret;        // ok
}
```

Do we really want these to be different? I view it as a bug, not a feature, that we treat them differently.

2. What is the purpose of `explicit`?

There seem to be at least two views about the purpose of `explicit`:

- Is `explicit` supposed to distinguish between copy and non-copy initialization?
- Is `explicit` there to prevent surprises by requiring the user to write more?

My opinion is that the purpose of `explicit` is best answered by Bjarne, who has opined that the “more letters means fewer mistakes” view is not always correct. However, that view may be the root of some people’s discomfort with this proposal.

Proposed Wording

Changes to 6.6.3/2 [stmt.return]:

> … A return statement with an expression of non-void type can be used only in functions returning a value~~; the value of the expression is returned to the caller of the function. The value of the expression is implicitly converted to the return type of the function in which it appears~~. A return statement in a function that returns a value initializes the returned object or reference from the expression or braced-init-list as specified in 8.5. A return statement can involve the construction and copy or move of a temporary object (12.2). [*Note*: A copy or move operation associated with a return statement may be elided or considered as an rvalue for the purpose of overload resolution in selecting a constructor (12.8). —*end note*] ~~A return statement with a braced-init-list initializes the returned object or reference by copy-list-initialization (8.5.4) from the specified initializer list.~~

In 8.5/15, change "function return" to "returning an expression from a function, returning a *braced-init-list* from a function with a reference return type (6.6.3)".

In 8.5/16, add "returning a *braced-init-list* from a function with an object return type (6.6.3)" to the list.

Q&A

This section captures highlights of discussion so far, mainly from the April reflector thread, the Portland and Rapperswil EWG discussion, the Rapperswil LEWG discussion, and Howard Hinnant’s paper N4094.

Should we then allow it symmetrically for parameters? No.

Mike Miller:

> How do you feel about something like
>
> ```
> void g(Type2);
> void f() {
>     g( {2} );    // Direct initialization?
> }
> ```
>
> I think I’d be less uncomfortable with your suggested "`return {2};`” being direct initialization if the same syntax applied to argument passing as well as value return.

This was part of my original (unpublished) proposal in Portland. The concern in EWG discussion was that function calls in general are a fundamental case where you want to disable the otherwise-implicit conversions from “my type” to “the other developer’s type,” such as not wanting `{100}` to turn into a `vector`.

The parameter and `return` cases really are not the same. Unlike `return` statements, with parameters:

- The authors of `f` and `g` are not the same, and the code is not local. With `return` the author is the same and the code is local.
- The function `g` could be overloaded, which could result in ambiguity. This cannot happen with `return`.
- As Bjarne said in Portland, ‘if you don’t know the return type of your function, your function is too long, and there are more arguments for the return case than the argument case.’

Should it be only for braced-init-lists? Yes.

Jonathan Wakely:

> I don't think we want to do this for all return statements, but maybe we could safely do it for return statements with a braced-init-list. (Otherwise we'd break at least `std::is_convertible`, and I don’t know what else.)

Is this redundancy coming up as an issue with our own proposals? Yes.

Gabriel Dos Reis gave this example and opinion:

> I would feel less miserable if
>
> ```
>   return { p };
> ```
>
> suffices and the language rules do not force me to have to repeat the return type that I just wrote before starting the body of the function—and no, I don't want constructors to be discriminated against based on `explicit` in this context. In case you wonder, see Appendix C (page 59) of
>
> http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2007/n2215.pdf
>
> In practice, the verbosity really is getting in the way of good codes.
>
> 3+ pages of function body isn’t a good reason to punish good codes.

What about `vector<int>` and `return {2}`? It’s unchanged.

Some have asked how the proposal affects this example:

```
vector<int> f() { return {2}; }  // one element or two?
```

In this proposal, the meaning of this code does not change.

Note: The general answer to what constructor is to be called for

```
X f() { return {expr}; }
```

is that it’s the same one that’s called for

```
X x{expr};
```

What about `vector<vector<int>>` and `return {2,2,2}`? It's unchanged.

Some have asked how the proposal affects this example:

```
vector<vector<int>> f() { return {2,2,2}; }
```

In this proposal, the meaning of this code does not change. It has the same meaning as

```
vector<vector<int>> x{2,2,2};
```

which is illegal.

What about `tuple<explicit_ctor>`? It's the same as initializing a local.

LEWG discussion in Rapperswil and N4094 considered the example of returning a `tuple` where one or more of the types has an explicit constructor. Since the `tuple` is an aggregate of individual values, this case is basically no different from returning each of those values, except that people will more naturally write `{}` in the return statement syntax so it's more possible to write it as a typo. Also, there are changes being proposed to `tuple`, and there was a question about those changes would affect this proposal.

The short answer, whatever `tuple`'s semantics are made to be, is that

```
tuple<explicit_ctor> f(){ return {expr}; }
```

would call the same constructor as

```
tuple<explicit_ctor> x{expr};
```

My view is that this too comes down the same core question: Distance. Does the return type being not on the same line make an important difference so as to feahave different syntax for local variable initialization and (I argue still "local") return type initialization?

```
tuple<explicit_ctor_type> f() {
    tuple<explicit_ctor_type> ret{value}; // ok
    return ret;                           // ok
    return {value};       // error today, yet a trivial refactoring
}
```

What about N4094's "The Counter-Example" example? It's no different.

Howard Hinnant, in N4094:

```
std::chrono::nanoseconds
stons(std::string const& str)
{
    using namespace std::chrono;
    auto errno_save = errno;
    errno = 0;
    char* endptr;
    auto count = std::strtoll(str.data(), &endptr, 10);
    std::swap(errno, errno_save);
    if (errno_save != 0)
        throw std::runtime_error("error parsing duration");
    if (str.data() < endptr && endptr < str.data() + str.size())
    {
        switch (*endptr)
        {
        case 'h':
            return hours{count};
        case 'm':
            if (endptr + 1 < str.data() + str.size())
            {
                switch (endptr[1])
                {
                case 'i':
                    if (endptr + 2 < str.data() + str.size() && endptr[2]=='n')
                        return {count};
                    break;
                case 's':
                    return milliseconds{count};
                }
            }
            break;
        case 's':
            return seconds{count};
        case 'u':
            if (endptr + 1 < str.data() + str.size() && endptr[1] == 's')
                return microseconds{count};
            break;
        case 'n':
            if (endptr + 1 < str.data() + str.size() && endptr[1] == 's')
                return nanoseconds{count};
            break;
        }
    }
    throw std::runtime_error("error parsing duration");
}
```

[…] There is a careless type-o / bug in `stons`. …

I have discussed this example with Howard, and my impression is that this too comes down to the same core question: Distance. Does the return type being not on the same line make an important difference so as to have different syntax for local variable initial and (I argue still "local") return type initialization?

```
chrono::nanoseconds f() {
    chrono::nanoseconds ret{count}; // ok
    return ret;                     // ok
    return {count};     // error today, yet a trivial refactoring
}
```

Besides that, there are secondary reasons I don't find this example convincing. First, this code demonstrates a rare case, namely multiple returns each doing a different explicit conversion. Second, this bug would be caught with the first unit test that exercised the return statement, and so isn't subtle. Third, this example was originally posted as a counterexample to an earlier version of the proposal that would have allowed `return x;` (without braces) to invoke explicit constructors; in the earlier version of the above example, the offending line was `return count;`, and that might have been more compellingly called a careless typo because it would be easy to write, but with the change to `return {count};` I find it harder to view the `{}` as a careless typo even though the example uses `{}` in the other return statements.

What about N4094's "Another Example" example? It's no different.

N4094 concludes with an extended example of where a series of changes to code during maintenance, with each changes seeming reasonable in isolation, can nevertheless lead to mistakes.

Writing bugs, including bugs under maintenance, is always possible in an unbounded number of ways in any significant language or library design. This is not a problem specific to the proposed explicit return syntax, or significantly contributed to by the proposed explicit return syntax.

Further, even if this were a problem, the same issue would arise with the trivial refactoring of introducing a local variable for the return value (already noted earlier).

What about implicit transfer of ownership for returning a unique_ptr? It's no different.

Ville Voutilainen:

> The main problem in handling returns of braced-initializers as implicit is that that would allow
>
> ```
> unique_ptr<int> f() {return {new int{}};}
> ```
>
> which itself is innocent, but worse, it would allow
>
> ```
> unique_ptr<int> T::f() {return {member_pointer};}
> ```
>
> which is a silent transfer of ownership.

I don't see this as any different from the following existing "pitfall," where we just shrug and say "but that's what you said to do":

```
unique_ptr<int> p{member_pointer};
```

In both this case and the return case, the object's type is explicitly a `unique_ptr`, and my view is that you are expected to know the return type of your own function.

Interestingly, by coincidence I came across the following case independently just a few hours before Ville wrote the above… In entirely unrelated code responding to a reader's email question, I tried essentially that very example:

```
template<class T>
std::unique_ptr<T> make_a() {
   T* p = nullptr;
   legacy(&p);                    // wraps a legacy allocation function
   return p;                      // error now and under this proposal
}
```

and I was surprised that it didn't work. Instead, I'm forced to write:

```
   return std::unique_ptr<T>{p}; // compiles; redundancy is mandatory
```

I would like to be able to write simply

```
    return {p};
```

which I believe is reasonable "explicit initialization syntax and what else could I possibly have meant" code. Today, I get the most developer-infuriating kind of diagnostic (being able to spell out exactly what the developer has to redundantly write to make the code compile, yet forcing the developer to type it out), and I think that it's unreasonable that I be forced to repeat `std::unique_ptr<T>` here.

Say, couldn't you just use an auto return type to avoid that redundancy? No.

Stephan Lavavej:

> Now that we have `auto` return types, repeating `unique_ptr` is unnecessary:
>
> ```
> template<class T>
> auto make_a() {
>     T* p = nullptr;
>     legacy(&p);
>     return std::unique_ptr<T>(p); // explicit acquisition of ownership
> }
> ```

This is not a general solution for two reasons that go beyond this specific example.

1. It isn't a legal option for separately compiled functions.
2. It doesn't remove the redundancy if there are multiple `return` statements.

Also, this actually documents yet another reason why the status quo is inconsistent, namely that the status quo allows deduction to work in all cases for:

```
auto test() {
   ...
   return T{x};          // ok
}
```

but does not support the inverse formulation in all cases, namely

```
T test() {
   ...
   return {x};           // ok sometimes, error other times
}
```

and that is inconsistent.

But, separately from this proposal, we could also enable auto on out-of-line definitions, right? Yes.

Coda: Stephan responded:

> I actually find it surprising that a normal declaration can't be followed by an `auto` definition with the same deduced type.

I agree that would be another nice EWG discussion, and I feel generally supportive of such a (separate) proposal, but it would still only address #1 above so it isn't a solution for this paper's issue.