Document #: N4029

Date: 2014-05-23

Reply to: Herb Sutter
hsutter@microsoft.com

Let `return` Be Direct and `explicit`

Herb Sutter

This paper addresses EWG issue #114.

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?

This paper proposes also permitting explicit conversions from the `return`'s expression to the return type, and generally viewing return-by-value as direct initialization of the returned value.

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. 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 T{expr};` does not add value, and is entirely 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_braces() {
    Type1 local1{1}; // ok
    return {1};      // ok
}
```

```
Type1 f1_parens() {
    Type1 local1(1); // ok
    return 1;        // ok
    return (1);      // ok
}

Type2 f2_braces() {
    Type2 local2{2}; // ok
    return {2};      // error, message could be “you must write Type2{2}”
    return Type2{2}; // ok but always redundant (no other type it could be)
}

Type2 f2_parens() {
    Type2 local2(2); // ok
    return 2;        // error, message could be “you must write Type2(2)”
    return (2);      // error, message could be “you must write Type2(2)”
    return Type2(2); // ok but always redundant (no other type it could be)
}
```

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.”

Proposed Resolution

Option 1: Allow explicit conversions to be considered (possibly smaller wording change)

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 ~~im~~explicitly converted to the return type of the function in which it appears. 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 by direct-list-initialization, or the returned reference ~~to be returned from the function~~ by copy-list-initialization (8.5.4), from the specified initializer list. …

Option 2: Use direct initialization (possibly more consistent)

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. If the function returns an

object by value, the return value is initialized from the expression by direct-initialization or direct-list-initialization. If the function returns a reference, t~~T~~he value of the expression is implicitly converted to the return type of the function ~~in which it appears. A return statement~~; this 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 by direct-list-initialization, or the returned reference ~~to be returned from the function~~ by copy-list-initialization (8.5.4), from the specified initializer list. …

Changes to 8.5/16 [dcl.init]:

The initialization that occurs in the forms

```
T x(a);

T x{a};
```

as well as in new expressions (5.3.4), `static_cast` expressions (5.2.9), functional notation type conversions (5.2.3), return by value expressions (6.6.3), and base and member initializers (12.6.2) is called *direct-initialization*.

Q&A

This section captures highlights of discussion so far, mainly from the April reflector thread and the Portland EWG wiki notes.

Q: Should we then allow it symmetrically for parameters? A: 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 Stroustrup said (from the EWG notes), 'if you don't know the return type of your function, your function is too long, and that there are more arguments for the return case than the argument case.'

Q: Should it be only for braced-init-lists? A: No.

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.)

The original proposal was to make braced-init-lists be the thing that would be considered "an explicit syntax." In Portland the EWG was persuasive that the key thing is the special nature of the `return` statement, not the special nature of the braced-init-list.

Also, the author of the `return` statement is the same as the author of the return type, and I think it would be inconsistent to allow

```
return {2};
```

and

```
int x{2};
return {x}; // ok if we made the braced-init-list the special thing
```

but not

```
int x{2};
return x;   // not ok if we made the braced-init-list the special thing
```

That would be a partial improvement, but it just moves the inconsistency around.

Q: What about implicit transfer of ownership for returning a unique_ptr? A: It's okay.

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.

This objection would apply also to the current revised proposal.

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 as Stroustrup said in the Portland EWG notes, you are expected to know the return type of your own function.

Speaking of which, code like Ville's above does come up in real life, because 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
}
```

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

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

I believe `return p;` is reasonable "what else could I possibly have meant" code, with 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.

Q: Couldn't you just use an auto return type to avoid the redundancy? A: 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
> }
> ```

No, for two reasons.

1. This isn't a legal option for separately compiled functions; deducing the return type requires the body to be present. (My example happened to be a template and therefore living in a header anyway, but not all functions are like that.)
2. It doesn't remove the redundancy if there are multiple `return` statements and you still have to say the type redundantly over and over again multiple times repeatedly. (My example happened to have one `return` statement, but not all functions are like that.)

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.

Coda: STL 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.

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

Gabriel Dos Reis:

> 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.

Agreed.