Doc. no. P0577R0
Date 2017-02-02
Reply-to Zhihao Yuan <zy@miator.net>
Audience Evolution Working Group

Keep that Temporary!

```
register lock_guard(mtx_);
string_view s = register to_string(42);
```

We propose *register-expression* to grant the temporary objects scope lifetimes.

Motivation

Lifetime extension that anyone can make use of. Prior work[1] on generalized lifetime extension uses sophisticated notations at the library side, but users cannot benefit from it when they simply don't own the library code if the library is not updated with those notations, or when the users and the library don't share the same opinion on whether a return value should be given extended lifetime. We want be able to freely choose when to extend the lifetime of a temporary regardless of the general considerations.

To eliminate the need of declaring a variable only for its destructor. EWG 35[2] attempts to solve this issue by generating uniquely named variables with some placeholder identifier:

```
auto φ = scope_exit(…);
```

But consider

```
auto temp = unique_ptr(…);
FILE* fp = temp.get();
```

The is a legitimate "declaring a variable only for its destructor", so the issue remains unsolved.

Introduction

Observing that in the "wrong way" of writing RAII,

```
scope_exit(…);
```

, the prvalue has been materialized into a temporary object, everything we physically need are physically there. An explicit lifetime extension notation is enough to get this work,

```
__extend_me scope_exit(…);
```

, no matter in which form it appears:[1]

```
FILE* fp = (__extend_me unique_ptr(…)).get();
```

Now we come to the notation we are proposing – `register`,

```
register scope_exit(…);
```

, reads, "register the temporary on the scope". We believe it will make more sense every time you want to register a temporary on the scope:[2]

```
auto t = (register (a + b).eval()).transpose();
```

[1] I'm not saying that this should be the style that we program in, but imagine a try_lock_guard which returns you a bool, or RAII wrappers returning error_code – it opens lots of possibilities.
[2] Example taken and modified from Eigen[3].

Design Decisions

Change the value category to lvalue after extending the lifetime. Let's take a closer look at the lifetime extension by reference binding in the standard:

```
auto&& r = to_string(42);
```

We always understand reference binding as the cause of lifetime extension. But if we forget about cause and effect and consider lifetime extension as a whole, we may notice that the interface of lifetime extension is always lvalue, like the `r` above. It's not a coincidence, because lifetime extension only makes sense when the result is an lvalue, because only lvalue can describe the lifetime and the expectation of a temporary with extended lifetime.

One noticeable benefit of this decision is that, for those library facilities who `= delete` their rvalue reference overloads to prevent accidentally binding to temporaries, they work with register expressions out of the box.

No lifetime extension for xvalue. The authors evaluated each kind of xvalue expressions, and concluded that there is no single efficient mechanism to ensure all xvalues' lifetimes, because xvalues are "(maybe) eXpiring" by nature. On the other hand, as shown in the following examples,

Imaginary xvalue lifetime extension syntax	Achieving similar effects with prvalue lifetime extension
`register f(g())`	`f(register g())`
`register X().n`	`(register X()).n`

, prvalue lifetime extension can be more straightforward and more efficient.

Be quiet when the notation has no effect. We want this facility to be friendly in generic programming context, where value category of the same expression may be dependent. So instead of triggering an error when a register expression is not extending the lifetime, evaluating it is as-if evaluating its operand. Note that this doesn't mean that such a register expression can be syntactically replaced by its operand, for example, for `int i;`, `decltype(register i)` must be `int&` rather than `int` as `decltype(i)` would answer.

Only appears in block scope. We ban register expression under 3 situations:

1. The expression will appear in one scope, but evaluate in another. This includes default member initializers, default arguments, and *ctor-initializer*s. They make the program less explicit, and their effects confuse even experts. The authors investigated and understood what every one of them does, then banned register expression from appearing in these contexts.
2. The lifetime of the temporary will extend to the end of the program. This includes namespace scope and initializers for static data members. Global RAII, or global temporary objects in general, have more problems then what register expression can solve. We plan to look at this part after the Modules TS being applied.
3. There is no scope to benefit from lifetime extension, such as enumeration scope.

To sum this up, block scope is the only one we allow for now.

The syntax models ***throw-expression*****.** We determined the operator precedence of `register` as follows:

First, adding `register` keyword to the subexpressions of a comma-separated expression shouldn't change its meaning,

Before	After
`a, b, c`	`a, register b, c`

, so `register` should be given higher precedence comparing to the comma operator.

Second, this precedence cannot be too high – the only unary operator that requires a space between the operator and an *id-expression* operand is `sizeof`,

sizeof	register
`sizeof a + b`	`register a + b`

, but it's more likely for people to agree on `sizeof`'s intention. To limit the confusion, we decided to place the precedence of `register` right below all binary operators (except comma) and the ternary operator,

```
register a ? b : c
```

, where a conditional lifetime extension makes more sense than a lifetime-extended condition. And that places *register-expression* at where *throw-expression* lives in BNF.

What It Is

Register expression is a callsite lifetime extension.

It is designed for letting any users freely choose when to use it. It has no impact on the callee-side, making it forward-compatible with any callee-side lifetime extension and/or diagnosis mechanism we may come up with in the future.

Register expression is a prvalue-to-lvalue cast.

Temporary materialization conversion is an implicit conversion from prvalue to xvalue, while register expression is an explicit conversion from prvalue to lvalue, i.e. a cast. They both materialize temporary objects, but give different value categories.

And What It's Not

Register expression is NOT a generalized lvalue cast.

Because it does not turn xvalue into lvalue. A longer explanation is, after observing that the `register` keyword works like a type function,

$$register : lvalue \to lvalue$$
$$register : prvalue \to lvalue$$

, it's tempting to expect register expression always answering lvalue, but that is not what register expression does. Register expression does not extend lifetimes for xvalues, so it will be a safety hole in the language if we decide to only perform the casts, and that is why we decided to keep "`register` *xvalue-expr*" as safe as *xvalue-expr* alone.

Register expression is NOT a declaration.

It is tempting to understand register expression as declaring a hidden variable with a unique name. But one must keep in mind that, in any context, adding the `register` keywords to subexpressions does not affect the order of evaluation, so there is no room to think about point of declaration. About the right way to understand register expression, see the title of this paper.

Practical Considerations

Compilers need to diagnose the uses of the `register` *storage-class-specifier* even after its removal, essentially supporting it. *register-expression* is not a declaration.

If a compiler can distinguish	, of course it can distinguish

`int i = 3;`
`i = 3;`
	`register int i = 3;`
	`register i = 3;`

What if users unnecessarily add `register` keywords? Users can unnecessarily factor out subexpressions into variable definitions as well. On the other hand, when adding `register` is really a regression, i.e, preventing copy elision, that particular situation is conceptually a prvalue-to-lvalue cast followed by an lvalue-to-rvalue conversion, so a compiler can easily diagnose it.

`T g();`
`void f(T);`	

`auto s = g();`
`f(s);`
	`f(register g());` *// warning*

Technical Description

assignment-expression:
    conditional-expression
    logical-or-expression assignment-operator initializer-clause
    throw-expression
    register-expression

register-expression:
    `register` *assignment-expression*

A register expression shall appear only in block scope (3.3.3).

Let *E* be the *assignment-expression* in a *register-expression*. If *E* is a prvalue expression of type `T` other than *cv* `void`, the register expression is an lvalue expression of type `T`, evaluating the register expression initializes a temporary object (12.2) of type `T` from *E* by evaluating *E* with the temporary object as its result object, and produces an lvalue denoting the temporary object; if the initialization is ill-formed, the program is ill-formed. Otherwise, the register expression is semantically equivalent to *E*. The lifetime of the temporary object extends to the end of the innermost enclosing scope where the register expression is evaluated in.

[Example: Given

```
struct A { int i; };
void f();
```

, `decltype(register A().i)` is `int&&` and `decltype(register f())` is `void`, while `decltype(register A())` is `A&`, allowing

```
int& r = (register A()).i;
r = 3;          // OK, assign 3 to t.i where t is the temporary object
                // introduced by register
```

– end example]

Acknowledgements

Thanks Tim Shen for cleaning up the design and reviewing this paper.

References

[1] David Krauss, N4221 *Generalized lifetime extension.*
http://www.open-std.org/JTC1/SC22/WG21/docs/papers/2014/n4221.pdf

[2] Jeffrey Yasskin, EWG 35 *Some concise way to generate a unique, unused variable name.*
https://cplusplus.github.io/EWG/ewg-active.html#35

[3] Bug 505 *Assert if temporary objects that are still referred to get destructed (was: Misbehaving Product on C++11)*
http://eigen.tuxfamily.org/bz/show_bug.cgi?id=505