Doc. no. D0577R1
Date 2018-07-09
Reply-to Zhihao Yuan <zy@miator.net>
Audience Evolution Working Group

Kept-value statement for guard objects

```
register lock_guard(a_mutex);
```

Motivation

Get rid of the macros represented by `BOOST_SCOPE_EXIT` in modern code.

Introduction

Currently, guard idioms such as `scope_exit` require defining variables to work, but the users need not refer to the guard objects. In such a case, inventing meaningless names for those variables makes the experience of using RAII likes recognizing a workaround rather than using a natural part of the language:

```
lock_guard lk(a_mutex);
```

A common practice is to wrap the declaration into a macro. Compare the above to the following "wrong way" of using guards:

```
lock_guard(a_mutex);
```

Both constructor and destructor have been called, just the destructor is called too early for our uses because this is a discarded-value expression. An explicit notation is enough to turn it into a "kept-value expression,"

```
__acquire lock_guard(a_mutex);
```

And the notation we propose here is "`register`,"

```
register lock_guard(a_mutex);
```

which reads "register the lock_guard to the scope."

Design Decisions

No traps. The following syntax

```
register lock_guard(a_mutex), lock_guard(a_second_mutex);
```

must not

lock `a_mutex`
lock `a_second_mutex`
unlock `a_mutex`
unlock `a_second_mutex`

The proposed resolution makes it simply not parsed. We could make it work as intended.

Do not detect overuses. We could say that registering an object with a trivial destructor makes the program ill-formed. But this can hardly be intentional. We consider this to be a QoI issue. It is worth noticing that registering *cv* `void` is intended to work as it happens in generic programming.

Appear in useful contexts. Global RAII objects that you cannot refer to are not comparable to those in block scope regarding the usefulness, so we are comfortable with the decision to make the proposed facility a *statement*. The following code can also benefit from the proposed semantics,

```
for (register lock_guard(a_mutex); …) {
    // locked scope
}
```

So the grammar should be available in an *init-statement* as well.

Alternatives

The lambda workaround

Wrapping the guarded code into a lambda serves the purpose:

```
lock_guard(a_mutex), [&] {
    // locked scope
}();
```

But first, it introduces a nested scope while saving such a level of *compound-statement* is sometimes being recognized as an advantage of RAII comparing to similar facilities in other languages. Second, it can hardly break out. At the time one can write range-based for, he or she will not write `std::for_each`, same reason.

Nameless variables

By observing that the name is unnecessary to the purpose of using an object's constructor and destructor, [P1110R0] proposes nameless variables. However, this is still a workaround because there is a gap between the means and the purpose – the purpose is to use an object's constructor and destructor, while the means is to define a variable. "[...] the usual workaround involves macros, which we claim we'd like to eliminate. '__' is a solution that helps to write the macro, but it isn't a solution for eliminating the macro." (Nevin).

Also, there are many ways to qualify a name even when the name intends to work as a placeholder:

```
auto &&__ = scope_exist(…);
auto *__ = scope_exist(…);
```

Not all of them make sense to the guard idioms, and some of them further detract from the readability.

Callee-side solutions

It is possible to make

```
lock_guard(a_mutex);
```

work as intended by introducing language features such as a notation on the (lock_guard) class, a special member operator, or a destructor overload.

But first we need to be careful about not to break the exception propagation graph, second, none of these helps existing libraries. And during the transition, mixing old style and new style libraries may lead to real confusions.

Bikeshedding

The author has no interests to defend the choice of the keyword.

Technical Description

Expression statements have two forms:

expression-statement:
expression$_{opt}$ `;`
`register` *assignment-expression* `;`

The first form is a discarded-value expression (8.2) [...]

Let *E* be the *assignment-expression* in the second form. If *E* is a prvalue expression of type `T` other than *cv* `void`, executing the statement initializes a temporary object (12.2) of type `T` from *E* by evaluating *E*; if the initialization is ill-formed, the program is ill-formed. The lifetime of the temporary object extends to the end of the innermost enclosing scope of the statement. Otherwise, the statement is equivalent to the first form with *E* as the *expression*.