P0052R2 - Generic Scope Guard and RAII Wrapper for the Standard Library

Peter Sommerlad and Andrew L. Sandoval

2016-02-12

Document Number: P0052R2	(update of N4189, N3949, N3830, N3677)
Date:	2016-02-12
Project:	Programming Language C++
Audience:	LWG/LEWG

1 History

1.1 Changes from P0052R1

The Jacksonville LEWG, especially Eric Niebler gave splendid input in how to improve the classes in this paper. I (Peter) follow Eric's design in specifying scope_exit as well as unique_resource in a more general way.

- Provide `scope_fail` and `scope_success` as classes. However, we may even hide these types and just provide the factories.
- safe guard all classes against construction errors, i.e., failing to copy the deleter/exit-function, by calling the passed argument in the case of an exception, except for scope_success.
- relax the requirements for the template arguments.

1.2 Changes from P0052R0

In Kona LWG gave a lot of feedback and especially expressed the desire to simplify the constructors and specification by only allowing *nothrow-copyable* `RESOURCE` and `DELETER` types. If a reference is required, because they aren't, users are encouraged to pass a `std::ref/std::cref` wrapper to the factory function instead.

- Simplified constructor specifications by restricting on nothrow copyable types. Facility is intended for simple types anyway. It also avoids the problem of using a type-erased `std::function` object as the deleter, because it could throw on copy.

- Add some motivation again, to ease review and provide reason for specific API issues.
- Make "Alexandrescu's" "declarative" scope exit variation employing `uncaught_exceptions()` counter optional factories to chose or not.
- propose to make it available for standalone implementations and add the header `<scope>` to corresponding tables.
- editorial adjustments
- re-established `operator*` for `unique_resource`.
- overload of `make_unique_resource` to handle `reference_wrapper` for resources. No overload for reference-wrapped deleter functions is required, because `reference_wrapper` provides the call forwarding.

1.3 Changes from N4189

- Attempt to address LWG specification issues from Cologne (only learned about those in the week before the deadline from Ville, so not all might be covered).
  - specify that the exit function must be either no-throw copy-constructible, or no-throw move-constructible, or held by reference. Stole the wording and implementation from unique_ptr's deleter ctors.
  - put both classes in single header `<scope>`
  - specify factory functions for Alexandrescu's 3 scope exit cases for `scope_exit`. Deliberately did't provide similar things for `unique_resource`.
- remove lengthy motivation and example code, to make paper easier digestible.
- Corrections based on committee feedback in Urbana and Cologne.

1.4 Changes from N3949

- renamed `scope_guard` to `scope_exit` and the factory to `make_scope_exit`. Reason for make_ is to teach users to save the result in a local variable instead of just have a temporary that gets destroyed immediately. Similarly for unique resources, `unique_resource`, `make_unique_resource` and `make_unique_resource_checked`.
- renamed editorially `scope_exit::deleter` to `scope_exit::exit_function`.
- changed the factories to use forwarding for the `deleter`/`exit_function` but not deduce a reference.
- get rid of `invoke`'s parameter and rename it to `reset()` and provide a `noexcept` specification for it.

1.5 Changes from N3830

- rename to `unique_resource_t` and factory to `unique_resource`, resp. `unique_resource_checked`
- provide scope guard functionality through type `scope_guard_t` and `scope_guard` factory
- remove multiple-argument case in favor of simpler interface, lambda can deal with complicated release APIs requiring multiple arguments.
- make function/functor position the last argument of the factories for lambda-friendliness.

1.6 Changes from N3677

- Replace all 4 proposed classes with a single class covering all use cases, using variadic templates, as determined in the Fall 2013 LEWG meeting.
- The conscious decision was made to name the factory functions without "make", because they actually do not allocate any resources, like `std::make_unique` or `std::make_shared` do

2 Introduction

The Standard Template Library provides RAII (resource acquisition is initialization) classes for managing pointer types, such as `std::unique_ptr` and `std::shared_ptr`. This proposal seeks to add a two generic RAII wrappers classes which tie zero or one resource to a clean-up/completion routine which is bound by scope, ensuring execution at scope exit (as the object is destroyed) unless released early or in the case of a single resource: executed early or returned by moving its value.

3 Acknowledgements

- This proposal incorporates what Andrej Alexandrescu described as scope_guard long ago and explained again at C++ Now 2012 ().
- This proposal would not have been possible without the impressive work of Peter Sommerlad who produced the sample implementation during the Fall 2013 committee meetings in Chicago. Peter took what Andrew Sandoval produced for N3677 and demonstrated the possibility of using C++14 features to make a single, general purpose RAII wrapper capable of fulfilling all of the needs presented by the original 4 classes (from N3677) with none of the compromises.

- Gratitude is also owed to members of the LEWG participating in the Fall 2015(Kona),Fall 2014(Urbana), February 2014 (Issaquah) and Fall 2013 (Chicago) meeting for their support, encouragement, and suggestions that have led to this proposal.

- Special thanks and recognition goes to OpenSpan, Inc. (http://www.openspan.com) for supporting the production of this proposal, and for sponsoring Andrew L. Sandoval's first proposal (N3677) and the trip to Chicago for the Fall 2013 LEWG meeting. *Note: this version abandons the over-generic version from N3830 and comes back to two classes with one or no resource to be managed.*

- Thanks also to members of the mailing lists who gave feedback. Especially Zhihao Yuan, and Ville Voutilainen.

- Special thanks to Daniel Krügler for his deliberate review of the draft version of this paper (D3949).

4 Motivation

While `std::unique_ptr` can be (mis-)used to keep track of general handle types with a user-specified deleter it can become tedious and error prone. Further argumentation can be found in previous papers. Here are two examples using `<cstdio>`'s `FILE *` and POSIX`<fcntl.h>`'s and `<unistd.h>`'s `int` file handles. See the following code examples on using `unique_resource` with `int` and `FILE *` handle types.

Both examples motivate the use case of the automatic conversion to use the return value of the factory as if it was the handle.

```
void demonstrate_unique_resource_with_stdio() {
  const std::string filename = "hello.txt";
  {
    auto file=make_unique_resource(::fopen(filename.c_str(),"w"),&::fclose);
    ::fputs("Hello World!\n", file);
    ASSERT(file.get()!= NULL);
  }
  {
    std::ifstream input { filename };
    std::string line { };
    getline(input, line);
    ASSERT_EQUAL("Hello World!", line);
    getline(input, line);
    ASSERT(input.eof());
  }
  ::unlink(filename.c_str());
  {
    auto file = make_unique_resource_checked(::fopen("nonexistingfile.txt", "r"),
                (FILE*) NULL, &::fclose);
    ASSERT_EQUAL((FILE*)NULL, file.get());
  }
}
```

```
void demontrate_unique_resource_with_POSIX_IO() {
  const std::string filename = "./hello1.txt";
  {
    auto file=make_unique_resource(::open(filename.c_str(),
                    O_CREAT|O_RDWR|O_TRUNC,0666), &::close);

    ::write(file, "Hello World!\n", 12u);
    ASSERT(file.get() != -1);
  }
  {
    std::ifstream input { filename };
    std::string line { };
    getline(input, line);
    ASSERT_EQUAL("Hello World!", line);
    getline(input, line);
    ASSERT(input.eof());
  }
  ::unlink(filename.c_str());
  {
    auto file = make_unique_resource_checked(::open("nonexistingfile.txt",
                     O_RDONLY), -1, &::close);
    ASSERT_EQUAL(-1, file.get());
  }

}
```

We refer to Andrej Alexandrescu's well-known many presentations as a motivation for scope_exit. Here is a brief example on how to use the 3 proposed factories. One is mandatory, the others are optional but address Andrej's examples.

```
void demo_scope_exit_fail_success(){
  std::ostringstream out{};
  auto lam=[&]{out << "called ";};
  try{
    auto v=make_scope_exit([&]{out << "always ";});
    auto w=make_scope_success([&]{out << "not ";}); // not called
    auto x=make_scope_fail(lam); // called
    throw 42;
  }catch(...){
    auto y=make_scope_fail([&]{out << "not ";}); // not called
    auto z=make_scope_success([&]{out << "handled";}); // called
  }
  ASSERT_EQUAL("called always handled",out.str());
}
```

5 Impact on the Standard

This proposal is a pure library extension. A new header, `<scope>` is proposed, but it does not require changes to any standard classes or functions. It does not require any changes in the core language, and it has been implemented in standard C++ conforming to C++14, resp. draft C++17. Depending on the timing of the acceptance of this proposal, it might go into library fundamentals TS under the namespace std::experimental or directly in the working paper of the standard, once it is open again for future additions.

6 Design Decisions

6.1 General Principles

The following general principles are formulated for `unique_resource`, and are valid for `scope_exit` correspondingly.

- Simplicity - Using `unique_resource` should be nearly as simple as using an unwrapped type. The generator functions, cast operator, and accessors all enable this.
- Transparency - It should be obvious from a glance what each instance of a `unique_resource` object does. By binding the resource to it's clean-up routine, the declaration of `unique_resource` makes its intention clear.
- Resource Conservation and Lifetime Management - Using `unique_resource` makes it possible to "allocate it and forget about it" in the sense that deallocation is always accounted for after the `unique_resource` has been initialized.
- Exception Safety - Exception unwinding is one of the primary reasons that `unique_resource` is needed. Nevertheless the goal is to introduce a new container that will not throw during construction of the `unique_resource` itself. However, there are no intentions to provide safeguards for piecemeal construction of resource and deleter. If either fails, no unique_resource will be created, because the factory function unique_resource will not be called. It is not possible to use `make_unique_resource()` factory with resource types, functors or lambda capture types as deleter objects where copying might throw.
- Flexibility - `unique_resource` is designed to be flexible, allowing the use of lambdas or existing functions for clean-up of resources.

6.2 Prior Implementations

Please see N3677 from the May 2013 mailing (or http://www.andrewlsandoval.com/scope_exit/) for the previously proposed solution and implementation. Discussion of N3677 in the (Chicago) Fall 2013 LEWG meeting led to the creation of `unique_resource` and `scope_exit` with the general agreement that such an implementation would be vastly superior to N3677 and would find favor with the LEWG. Professor Sommerlad produced the implementation backing this proposal during the days following that discussion.

N3677 has a more complete list of other prior implementations.

N3830 provided an alternative approach to allow an arbitrary number of resources which was abandoned due to LEWG feedback

The following issues have been discussed by LEWG already:

- *Should there be a companion class for sharing the resource `shared_resource` ? (Peter thinks no. Ville thinks it could be provided later anyway.)* LEWG: NO.
- *Should `scope_exit()` and `unique_resource::invoke()` guard against deleter functions that throw with `try deleter(); catch(...)` (as now) or not?* LEWG: NO, but provide noexcept in detail.
- *Does `scope_exit` need to be move-assignable?* LEWG: NO.

The following issues have been recommended by LWG already:

- Make it a facility available for free-standing implementations in a new header `<scope>` (`<utility>` doesn't work, because it is not available for free-standing implementations)

6.3 Open Issues to be Discussed by LEWG

- Should we make the regular constructor of the scope_exit templates private and friend the factory function only? This could prohibit the use as class members, which might sneakily be used to create "destructor" functionality by not writing a destructor.
- Should we provide factories `make_scope_success(ef)` and `make_scope_fail(ef)` to enable Alexandrescu's three scope-exiting modes?
- Even though LWG didn't like the conversion operator of `unique_resource`, the authors believe it is valuable to be able to use the wrapper like it was the original resource, thus easing integration of `unique_resource` into existing code (see examples above).

7 Technical Specifications

The following formulation is based on inclusion to the draft of the C++ standard. However, if it is decided to go into the Library Fundamentals TS, the position of the texts and the namespaces will have to be adapted accordingly, i.e., instead of namespace `std::` we suppose namespace `std::experimental::`.

7.1 Header

In section 17.6.1.1 Library contents [contents] add an entry to table 14 for the new header `<scope>`.

In section 17.6.1.3 Freestanding implementations [compliance] add an extra row to table 16 and in section [utilities.general] add the same extra row to table 44

Table 1: table 16 and table 44

	Subclause	Header
20.nn	Scope Guard Support	`<scope>`

7.2 Additional sections

Add a a new section to chapter 20 introducing the contents of the header `<scope>`.

7.3 Scope guard support [scope]

This subclause contains infrastructure for a generic scope guard and RAII (resource acquisition is initialization) resource wrapper.

Header `<scope>` synopsis

```
namespace std {
template <typename EF>
class scope_exit;
template <typename EF>
class scope_fail;
template <typename EF>
class scope_success;

template <typename EF>
scope_exit<EF> make_scope_exit(EF exit_function) ;
template <typename EF>
scope_fail<EF> make_scope_fail(EF exit_function) ;
template <typename EF>
scope_success<EF> make_scope_success(EF exit_function) ;

template<typename R,typename D>
class unique_resource;

template<typename R,typename D>
unique_resource<R, D>
make_unique_resource( R  r, D d) noexcept;
template<typename R,typename D, typename RI=R>
unique_resource<R, D>
make_unique_resource_checked(R r, RI invalid, D d) noexcept;

}
```

1 The header `<scope>` defines the class templates `scope_exit`, `scope_fail`, `scope_success`, `unique_resource` and the factory function templates `make_scope_exit()`, `make_scope_success()`, `make_scope_fail()`, `make_unique_resource()` and `make_unique_resource_checked()` to create their instances.

7.3.1 Class template `scope_exit` [scope.scope_exit]

```
template <typename EF>
class scope_exit {
public:
  explicit
  scope_exit(EF f) ;
  scope_exit(scope_exit&& rhs) ;
  ~scope_exit() ;
  void release() noexcept;

  scope_exit(const scope_exit&)=delete;
  scope_exit& operator=(const scope_exit&)=delete;
  scope_exit& operator=(scope_exit&&)=delete;
private:
  EF exit_function;     // exposition only
};
template <typename EF>
class scope_fail {
//@seebelow@
};
template <typename EF>
class scope_success {
//@seebelow@
};
```

1 [*Note:* `scope_exit` is meant to be a general-purpose scope guard that calls its exit function when a scope is exited. The class templates `scope_fail` and `scope_success` share the `scope_exit`'s interface, only the situation when the exit function is called differs. These latter two class templates memorize the value of `uncaught_exceptions()` on construction and in the case of `scope_fail` call the exit function on destruction, when `uncaught_exceptions()` at that time returns a greater value, in the case of `scope_success` when `uncaught_exceptions()` on destruction returns the same or a lesser value. — *end note*]

2 If template argument `EF` is not a reference type, `EF` shall satisfy the requirements of `Destructible` (Table 24).

3 The following specification only lists special cases for `scope_fail` and `scope_success`, when their semantic differs from `scope_exit`.

```
explicit
scope_exit(EF f) ;
```

4 *Effects:* Constructs a `scope_exit` object that will call `f()` on its destruction unless `release()` was called prior to that.

```
explicit
scope_fail(EF f) ;
```

5 *Effects:* Constructs a `scope_fail` object that will call `f()` on its destruction if its scope is left with a new exception, unless `release()` was called prior to that.

```
explicit
scope_success(EF f) ;
```

6 *Effects:* Constructs a `scope_success` object that will call `f()` on its destruction if its scope is left without an exception, unless `release()` was called prior to that.

```
scope_exit(scope_exit&& rhs) ;
scope_fail(scope_fail&& rhs) ;
scope_success(scope_success&& rhs) ;
```

7 *Effects:* Copies the release state from `rhs`, and sets `rhs` to the released state, preventing it from invoking its copy of `exit_function`. If `is_nothrow_move_constructible<EF>` move constructs otherwise copy constructs `exit_function` from `rhs.exit_function`. In case of an exception during the last operations, calls `rhs.exit_function()` if it would be called when `rhs` would have been destroyed without being moved from.

```
˜scope_exit();
˜scope_fail();
˜scope_success();
```

8 *Effects:* Calls `exit_function()` if the rules of the class given at its constructor specification ask for it, unless `release()` was previously called.

```
void release() noexcept;
```

9 Prevents `exit_function()` from being called on destruction.

7.3.2 `scope_exit` factory functions [scope.make_scope_exit]

1 The factory functions create `scope_exit`, `scope_fail`, and `scope_success` objects that run `exit_function` at scope exit under the following conditions unless `release()` was called on the returned object:

make_scope_exit always, if scope is exited

make_scope_fail if scope is exited by throwing an exception

make_scope_success if scope is exited without any exception

```
template <typename EF>
scope_exit<EF> make_scope_exit(EF exit_function) ;
```

2 The factory function creates a `scope_exit` object, that runs `exit_function` at scope exit unless `release()` was called before.

```
template <typename EF>
scope_fail<EF> make_scope_fail(EF exit_function) ;
```

3 The factory function creates a `scope_fail` object, that runs `exit_function` at scope exit from a new exception, unless `release()` was called before.

```
template <typename EF>
scope_success<EF> make_scope_success(EF exit_function) ;
```

4 The factory function creates a `scope_fail` object, that runs `exit_function` at scope exit from a new exception, unless `release()` was called before.

7.3.3 Unique resource wrapper [scope.unique_resource]

7.3.4 Class template `unique_resource` [scope.unique_resource.class]

```
template<typename R,typename D>
class unique_resource {
public:
  unique_resource(const R & r, const D & d) ;
  unique_resource(R && r, const D & d) ;
  unique_resource(const R & r, D && d) ;
  unique_resource(R && r, D && d) ;
  unique_resource(unique_resource&& rhs) ;
  unique_resource& operator=(unique_resource&& rhs) ;
  unique_resource& operator=(unique_resource const &)=delete;
  unique_resource(unique_resource const &)=delete;
  ~unique_resource();
```

```
    void swap(unique_resource &other);
    void reset();
    void reset(R r);
    void release() noexcept;
    R const & get() const noexcept;
    R operator->() const noexcept;
    see below operator*() const noexcept;
    const D & get_deleter() const noexcept;
  private:
    R resource; // exposition only
    D deleter; // exposition only
    bool execute_on_destruction; // exposition only
  };
```

1 [*Note:* `unique_resource` is meant to be a universal RAII wrapper for resource handles provided by an operating system or platform. Typically, such resource handles are of trivial type and come with a factory function and a clean-up or deleter function that do not throw exceptions. The clean-up function together with the result of the factory function is used to create a `unique_resource` variable, that on destruction will call the clean-up function. Access to the underlying resource handle is achieved through `get()` and in case of a pointer type resource through a set of convenience pointer operator functions. *— end note*]

2 The template argument `D` shall be a Destructible (Table 24) function object type (20.9), for which, given a value `d` of type `D` and a value `r` of type `R`, the expression `d(r)` is valid and does not throw an exception.

3 `R` shall be a Destructible (Table 24) type.

```
unique_resource(const R & r, const D & d) ;
unique_resource(const R & r, D && d) ;
unique_resource(R && r, const D & d) ;
unique_resource(R && r, D && d) ;
```

4 *Effects:* constructs a `unique_resource` from `r` and `d`. If construction fails, calls `d(r)`.

5 *Postconditions:* `get() == r`. `get_deleter()` returns a reference to the stored function object `d`.

6 [*Note:* If `R` is a non-const lvalue reference type, only the latter two overloads are available. *— end note*]

```
unique_resource(unique_resource&& rhs) ;
```

7 *Effects:* Move constructs from the value `rhs`, then calls `rhs.release()`. If construction fails, as if `rhs.reset()`.

```
unique_resource& operator=(unique_resource&& rhs) ;
```

8 *Effects:* If `this == &rhs` no effect, otherwise `reset()`, then move assigns from `rhs`, then `rhs.rlease()`. If either fails, as if on the not-moved-from `rhs`: `rhs.reset()`.

`~unique_resource();`

9 *Effects:* `reset()`.

`void reset();`

10 *Effects:* Equivalent to

```
if (execute_on_destruction) {
  execute_on_destruction=false;
  get_deleter()(resource);
}
```

`void reset(R r) ;`

11 *Effects:* Equivalent to

```
reset();
resource = move(r);
execute_on_destruction = true;
```

If move-assignment of the resource fails, `get_deleter()(r)` on the original value of r.

`void release() noexcept;`

12 *Effects:* `execute_on_destruction = false`.

```
const R& get() const noexcept ;
R operator->() const noexcept ;
```

13 *Requires:* `operator->` is only available if
`is_pointer<R>::value && is_nothrow_copy_constructible<R>::value`
`&&(is_class<remove_pointer_t<R>>::value || is_union<remove_pointer_t<R>>::value)`
is `true`.

14 *Returns:* `resource`.

`see below operator*() const noexcept ;`

15 *Requires:* `operator*` is only available if `is_pointer<R>::value` is `true`.

16 *Returns:* `*resource`. [*Note:* The return type is equivalent to `add_lvalue_reference_t<remove_pointer_t<R>>`. — *end note*]

`const D & get_deleter() const noexcept;`

17 *Returns:* `deleter`

7.3.5 Factories for `unique_resource` [scope.make_unique_resource]

```
template<typename R,typename D>
unique_resource<decay_t<R>, decay_t<D>>
make_unique_resource( R && r, D && d) noexcept;
```

1 *Returns:* `forward<R>(r), forward<D>(d)`

```
template<typename R,typename D>
unique_resource<R&,D>
make_unique_resource( reference_wrapper<R> r, D d) noexcept;
```

2 *Returns:* `unique_resource<R&,D>(r.get(),d)`

3 [*Note:* There is no need to overload on `reference_wrapper` for the deleter. — *end note*]

```
template<typename R,typename D, typename S=R>
unique_resource<decay_t<R>,decay_t<D>>
make_unique_resource_checked(R&& r, S const & invalid, D && d ) noexcept;
```

4 *Requires:* If `S` is the same type as `R`, `R` shall be EqualityComparable(Table 17). Otherwise, the expression `r==invalid` shall be valid and return a value that is convertible to `bool`. 5 *Effects:* As if

```
    bool mustrelease = bool(r == invalid);
    auto ur= make_unique_resource(forward<R>(r), forward<D>(d));
    if(mustrelease) ur.release();
    return ur;
```

8 Appendix: Example Implementations

removed, see
https://github.com/PeterSommerlad/SC22WG21_Papers/tree/master/workspace/P0052_scope_exit/src